Towards Transparent Throughput Elasticity for IaaS Cloud Storage: Exploring the Benefits of Adaptive Block-Level Caching

Bogdan Nicolae, Pierre Riteau, Kate Keahey

Towards Transparent Throughput Elasticity for IaaS Cloud Storage: Exploring the Benefits of Adaptive Block-Level Caching

Bogdan Nicolae
IBM Research, Ireland
bogdan.nicolae@acm.org

Pierre Riteau
University of Chicago, USA
priteau@uchicago.edu

Kate Keahey
Argonne National Laboratory, USA
keahey@mcs.anl.gov

***Abstract*—Storage elasticity on IaaS clouds is a crucial feature in the age of data-intensive computing, especially when considering fluctuations of I/O throughput. This paper provides a transparent solution that automatically boosts I/O bandwidth during peaks for underlying virtual disks, effectively avoiding over-provisioning without performance loss. Our proposal relies on the idea of leveraging short-lived virtual disks of better performance characteristics (and thus more expensive) to act during peaks as a caching layer for the persistent virtual disks where the application data is stored. We show how this idea can be achieved efficiently at the block-device level, using a caching mechanism that leverages iterative behavior and learns from past experience. Furthermore, we introduce a performance and cost prediction methodology that can be used both independently to estimate in advance what trade-off between performance and cost is possible, as well as an optimization technique that enables better cache size selection to meet the desired performance level with minimal cost. We demonstrate the benefits of our proposal both for microbenchmarks and for two real-life applications using large-scale experiments.**

***Keywords*-cloud computing, storage elasticity, adaptive I/O, virtual disk, block-level caching, performance prediction, cost prediction, modeling**

I. Introduction

Elasticity (i.e., the ability to acquire and release resources on-demand as a response to changes of application requirements during runtime) is a key feature that drives the popularity of infrastructure clouds (Infrastructure-as-a-Service, or IaaS, clouds). To date, much effort has been dedicated to studying the elasticity of computational resources, which in the context of IaaS clouds is strongly related to the management of virtual machine (VM) instances [18], [19], [27]: when to add and terminate instances, how many and what type to choose, and so forth. *Elasticity of storage* has gained comparatively little attention, however, despite the fact that applications are becoming increasingly data-intensive and thus need cost-effective means to store and access data.

An important aspect of storage elasticity is the management of I/O access throughput. Traditional IaaS platforms offer little support to address this aspect: users have to manually provision raw virtual disks of predetermined capacity and performance characteristics (i.e., latency and throughput) that can be freely attached to and detached from VM instances (e.g. , Amazon Elastic Block Storage (EBS) [1]). Naturally, provisioning a slower virtual disk incurs lower costs when compared with using a faster disk; however, this comes at the expense of potentially degraded application performance because of slower I/O operations.

This trade-off has important consequences in the context of large-scale, distributed scientific applications that exhibit an iterative behavior. Such applications often interleave computationally intensive phases with I/O intensive phases. For example, a majority of high-performance computing (HPC) numerical simulations model the evolution of physical phenomena in time by using a bulk-synchronous approach. This involves a synchronization point at the end of each iteration in order to write intermediate output data about the simulation, as well as periodic checkpoints that are needed for a variety of tasks [22] such as migration, debugging, and minimizing the amount of lost computation in case of failures. Since many processes share the same storage (e.g., all processes on the same node share the same local disks), this behavior translates to periods of little I/O activity that are interleaved with periods of highly intensive I/O peaks.

Since time to solution is an important concern, users often over-provision faster virtual disks to achieve the best performance during I/O peaks and under-use this expensive throughput outside the I/O peaks. Since scientific applications tend to run in configurations that include a large number of VMs and virtual disks, this waste can quickly get multiplied by scale, prompting the need for an elastic solution.

This paper extends our previous work [26] where we introduced an elastic disk throughput solution that can deliver high performance during I/O peaks while minimizing costs related to storage. Our initial proposal focused on the idea of using small, short-lived, and fast virtual disks to temporarily boost the maximum achievable throughput during I/O peaks by acting as a caching layer for larger but slower virtual disks that are used as primary storage. We have shown how this approach can be efficiently achieved in a completely *transparent* fashion by exposing a specialized block device inside the guest operating system that hides all details of virtual disk management at the lowest level, effectively casting the throughput elasticity as a block-device caching problem where performance is complemented by cost considerations.

In this paper, we complement our previous work by ex-

ploring how to predict the performance and cost of running HPC applications that exhibit well-defined I/O behavior. This is a critical challenge, because running such applications at scale requires a massive amount of resources, which makes it important to know in advance whether it is feasible to obtain the desired results within a given deadline and cost.

Our contributions can be summarized as follows:

- We refine the concept of throughput elasticity, as well as a series of requirements, design considerations and a cost model. These enhancements enable our proposal to provide throughput elasticity in a transparent fashion at the block-device level. In particular, we describe a caching strategy that adapts to an iterative behavior and learns from the past experience in order to maximize the I/O boost during peaks, while minimizing the usage of fast virtual disks. (Section III-A)
- We introduce a novel methodology to predict the performance and cost of running large scale HPC applications that interleave computationally intensive phases with write intensive phases corresponding to checkpointing. We also show how these results can be used to optimize the cache size selection strategy in order to minimize the cost for a particular run within given performance degradation bounds. (Sections III-C and III-D)
- We show how to apply these design considerations in practice through a series of building blocks (and their corresponding implementation as a prototype) that run inside the VM instances of the users and interact with a typical IaaS cloud architecture. (Sections III-E and III-F)
- We evaluate our approach in a series of experiments conducted on dozens of nodes of the Shamrock experimental testbed, using both synthetic benchmarks and HPC real-life applications. In this context, we demonstrate large real-life reductions of storage cost at the expense of minimal performance overhead when compared with over-provisioning. We also validate our performance and cost prediction methodology against the real life results, showing high prediction accuracy. Finally, we apply our prediction proposal to these real-life HPC applications as an illustrative case study in order to demonstrate the pre-optimizations it can achieve. (Section IV)

II. Related work

Hybrid file systems [34], [31] and associated caching strategies [10], [37] have long been used to combine multiple devices of different types (e.g., SSDs, HDDs, nonvolatile memories). Generally, however, they are designed to use fixed storage resources available as physical hardware in order to improve access performance. Unlike our approach, such strategies are not concerned with being economical and minimizing resource usage. Furthermore, if the application needs a storage abstraction that is not file-based, then a file system introduces unnecessary overhead.

In the area of caching, Wang et. al. studied the problem of dynamically selecting the caching policy under varying workloads [35]. The caching framework selects a caching policy and reconfigures the storage system on the fly based on access traces gathered and analyzed during application runtime. An aspect of adaptation to access pattern is also explored by our previous work [24], [23], however, the focus is on scalable virtual disk on-demand image content delivery at large scale. Also with respect to caching strategies, an increasingly popular target is the newer generation of flash-based devices. For example, in [5], special hybrid trees are proposed to organize and manipulate intervals of cached writes. Although orthogonal to our own work, such efforts provide valuable insight in terms of how caching is handled on the host that exposes the virtual disks to the VMs, which may influence how to best leverage guest-level caching if additional information is available.

Moving upward in the storage stack at the virtualization level, several approaches aim to accelerate the throughput of virtual disks at hypervisor level or below. Jo et al. [13] proposed a hybrid virtual disk based on a combination of an SSD and an HDD. Contrarily to our solution, the SSD in their approach is read-only and used only to improve read performance to the template VM image. All write operations are sent to the HDD to avoid degrading the performance of flash storage. Using a fast device to cache both reads and writes from a read-only virtual disk snapshot is possible using copy-on-write and/or mirroring [30], [20], [21]. However, one of the disadvantages in this context is fragmentation, for which specialized strategies might be necessary [22].

How to virtualize bandwidth in a cloud environment was explored at various levels. The S-CAVE developers [17] propose to leverage the unique position of the hypervisor in order to efficiently share SSD caches between multiple VMs. Similarly, vPFS [38] introduces a bandwidth virtualization layer for parallel file systems that schedules parallel I/Os from different applications based on configurable policies. Unlike our approach, the focus in this context is bandwidth isolation between multiple clients, as opposed to elasticity.

Storage elasticity on IaaS clouds was explored at coarse granularity by Lim et al. [15] for multi-tier application services, with a focus on how to add and remove entire storage nodes and how to rebalance the data accordingly. Higher level service processing acceleration was described in [7]. This work introduces an elasticity aspect in the form of a series of algorithms to scale the cache system up during peak query times and back down afterwards to save costs. *LogBase* [33] is another elastic storage effort that employs a log-structured database system targeted at write-intensive workloads. Unlike our approach, the goal is to improve the write performance and simplify recovery. Chen et al. introduced *Walnut* [6], an object store that provides elasticity and high availability across Yahoo!'s data clouds and is specifically optimized for the data-intensive workloads observed in these clouds: Hadoop [36], MObStor (unstructured storage similar to [2]), PNUTS [8], and so forth. The main goal is sharing of hardware resources across hitherto siloed clouds of different types, offering greater potential for intelligent load balancing and efficient elastic operation, while simplifying the operational tasks related to

data storage.

Several approaches acknowledge the importance of estimating and leveraging cost and performance prediction for clouds. *Log2cloud* [28] is one such effort that uses established results from queueing network theory to predict the minimum VM cost of cloud deployments starting from existing application logs. Other research focuses on modeling specific applications and workloads, such as database queries [14]. Specifically in the area of storage, efforts such as ACIC [16] automatically search for optimized I/O system configurations, relying on machine learning models to perform black-box performance/cost predictions.

Our own previous work [25] focuses on storage elasticity from a different perspective: space utilization, aiming to adapt transparently to growing/shrinking data sizes by means of a POSIX-compatible file system that automatically adds and removes virtual disks accordingly. With respect to throughput elasticity, we have explored in our previous work [26] a generic approach that adapts to the I/O intensive phases of the application automatically, striking a good all-around trade-off between minimizing the waste caused by over-provisioning of throughput while at the same time minimizing the performance degradation.

This work also focuses on throughput elasticity, but from a complementary perspective: it contributes with a new performance and cost prediction methodology that enhances our previous work for well-defined I/O access patterns consisting of synchronized write-intensive checkpointing. This pattern is exhibited by a large class of HPC applications. The predictions can be used both independently to estimate in advance what trade-off between performance and cost is possible, as well as an optimization technique that enables better cache size selection to meet the desired performance level with minimal cost.

III. System design

In a nutshell, our proposal relies on a simple core idea: the use of small and short-lived virtual disks of high-throughput capability (with higher price per gigabyte) to *transparently* boost the I/O performance during peak utilization of slower (and cheaper per gigabyte) virtual disks that are continuously used by the application to accumulate persistent data. We use the former as a ephemeral caching device and the latter as a backing device. When the caching device is in operation, it acts as a read/write caching layer at the block level that uses an adaptive mechanism to asynchronously flush dirty blocks back to the backing device. Besides the technical challenges related to achieving transparency efficiently at the block level, the main challenge in this context is the focus on cost reduction, which brings a novel perspective to the otherwise well-studied caching domain. Several critical questions arise: How large should the virtual disk acting as a cache be? When and for how long do we need it? What caching strategy should we use?

To answer these questions, we introduce a series of design considerations formulated in response to the problem we study. We give a general description of these design considerations (Section III-A), show how to adopt them in a typical IaaS cloud (Section III-E), and briefly describe a prototype that implements our approach (Section III-F). We also propose a cost model to quantify the utilization of throughput (Section III-B), how to estimate the performance and cost based on this model (Section III-C) and how to use such estimations to optimize the cache size selection (Section III-D).

A. Design considerations

Our proposal relies on three key design principles:

1) Transparent block level caching: Storage is typically provisioned on IaaS clouds in the form of virtual disks that are created by using a predefined size and performance characteristics (i.e., throughput). Although the virtual disks can be freely attached to and detached from running VM instances, this degree of elasticity is hard to leverage directly at the application level in order to deal with fluctuating I/O throughput requirements: data would have to be constantly migrated to/from a slower/faster device, thereby generating high performance and cost overheads that are unacceptable if a large amount of data accumulates during runtime or if the fluctuations happen over short periods of time. Even if such an approach were feasible, applications often do not leverage virtual disks directly but rely on storage abstractions (e.g., a file system) that were not designed to add/remove disks on the fly. Thus, it is desirable to handle throughput elasticity in a *transparent* fashion at the lowest level.

In response to this need, we propose a solution that works at the *block level*. Specifically, we expose a block device in the guest operating system that replaces the virtual disk normally leveraged by users directly, using it as a *backing device* that all I/O is redirected to. When I/O throughput utilization rises above the utilization threshold (UT), a second, faster virtual disk (referred to as the *caching device*) is provisioned to act as a caching layer for the backing device, temporarily boosting I/O throughput until the threshold falls below UT. At this point, the data from the caching device is flushed to the backing device, and the caching device is removed, with I/O passing directly to the backing device again. Such an approach enables transparency not only from the user's perspective but also from the cloud provider's perspective: it works at the guest level and does not require changes to the virtualization infrastructure or provisioning model.

2) Adaptive flushing of dirty blocks: A solution that alternates between a fast and a slow virtual disk to achieve elasticity suffers from poor I/O performance and is unsustainable, because an increasingly larger set of accumulated data needs to be migrated between the two devices. On the other hand, a solution based on caching dramatically reduces the amount of data movements, because only the most recently used blocks are involved (we call these blocks "hot"). Even when considering only the "hot" blocks, however, the constant movement between the backing device and the caching device naturally steals bandwidth from both devices,

effectively limiting the potential to boost the I/O throughput at full capacity.

To address this issue, we propose to make the caching device act like a regular block-level read/write cache but with a custom dirty block commit strategy. More specifically, during a read operation, any requested blocks that are not already available on the caching device are first fetched from the backing device and written to the caching device by using an LRU (least recently used) eviction strategy. Then, the read operation is fully redirected to the caching device. In the case of a write operation, all dirty blocks are initially written to the cache only and are later committed to the backing device.

The strategy to commit dirty blocks works in two phases. In the first phase, we use a mechanism that closely resembles writeback and prioritizes the application I/O: it flushes dirty blocks asynchronously to the backing device only when spare bandwidth is available or when the caching device is full and needs to evict. Once the required I/O throughput drops to a level that the backing device is able to sustain on its own, a transition to the second phase is initiated, in which the priority is reversed: the flushing process proceeds at full speed at the expense of application I/O. At the same time, only reads are allowed from the caching device starting from this moment onwards, with writes bypassing the caching device and being redirected to the backing device. We refer to this two-phase strategy as "dynamic writeback."

Note that finding the right moment to reverse the priority is important: if it happens too soon, the application will suffer a performance penalty because of background flushes. If it happens too late, the caching device stays up longer than necessary and thus incurs extra costs. In order to deal with this issue, the decision of when to reverse the priority is based on a configurable amount of time ID (inactivity delay), which represents how much the application's I/O throughput needs to stay below UT before we reverse the flush priority. Since the flush process is prioritized after the reverse and writes bypass the caching device, eventually all dirty blocks will be committed to backing device. At this point, one can safely detach the caching device and remove its corresponding virtual disk.

3) Access-pattern-aware cache sizing: Using a caching device can be expensive: while it is active, the user is charged for *both* the caching device and the backing device. Thus, the caching device cannot be arbitrarily large for two reasons.

First, the flushing of dirty data does not happen instantly after the decision was made to proceed to the second phase of the commit strategy, which delays the moment when the caching device can be safely removed. Second, since dirty data tends to accumulate proportionally to the cache size, a large caching device is likely to cause a long flush delay. On the other hand, if the cache size is too small, flushing may be forced prematurely, limiting the potential to boost application I/O at full capacity. Thus, it is important to automatically optimize the cache size specifically for the access pattern of the application observed during the I/O-intensive phase.

To optimize the size of the caching device, we propose to leverage the predominantly repetitive I/O behavior of large-scale scientific applications in order to learn from the experience of the previous I/O-intensive phases for which a caching device was in use. More specifically, we start with a large cache size for the first time when an I/O boost is needed and then monitor the cache utilization. If the caching device was used only partially, then we decrease its size for the next I/O-intensive phase down to the size that was actually used. Similarly, if too much flushing was forced prematurely during the first phase of the commit strategy (in which application I/O is prioritized), we increase the cache size for the next I/O-intensive phase by the amount that had to be evicted.

B. Cost model

We assume a cost model that charges users for utilization at fine grain that can be as little as the order of seconds. This approach is already adopted in real-life: for instance, Google Compute Engine charges persistent disks at a granularity of seconds [11]. Since we are dealing with different types of virtual disks, we approximate a realistic cost by defining utilization as a function of both the size of the virtual disk and its throughput characteristics (i.e., reserved bandwidth).

Note that the utilization is based on reserved bandwidth, which relates to the high-end spectrum of cloud offerings such as the Object Storage offered by *IBM SoftLayer* [32] or provisioned Elastic Block Store (EBS) volumes offered by *Amazon* [1]. This is different from the more popular "classic EBS" model where users are not offered any bandwidth guarantees and are charged per IOPS instead. We justify the need to rely on reserved bandwidth because our primary target is tightly coupled HPC applications that run at large scale and are known for their susceptibility to *system noise amplification* [12] (i.e., the bulk-synchronous nature makes processes sensitive to delays that affect all other processes, which in our context means that a slow virtual disk attached to a VM instance causes slowdown of all other VM instances where the application processes are running). Thus, reasoning in terms of average throughput (as is the case of classic EBS volumes) is not feasible in our context.

To quantify the utilization in accordance with the requirements mentioned above, we introduce a metric called *adjusted storage accumulation*, which reflects the total cost that accumulates over a period of time t for a VM instance as a result of storage use. We assume that for every time unit of utilization, a virtual disk of size N and reserved bandwidth B incurs a cost of $N \cdot B$. Thus, if a single virtual disk is attached to a VM instance for the whole duration t, the total cost is $C(t) = B \cdot N \cdot t$. When using both a backing device and a caching device to implement our approach, the backing device will continuously contribute to the total cost, as in the case of a single virtual disk, whereas the caching device will contribute only while it is used and proportionally to its dynamically adjusted size. More formally, this approach can be expressed as follows.

$$C(t) = B_b \cdot N \cdot t + B_c \cdot \int_0^t M(x) \cdot dx$$

N and B_b are the size and bandwidth of the backing device, respectively; B_c is the bandwidth of the caching device; and $M(x)$ is the size of the caching device at a given moment x (0 if the caching device is not used at that moment). We express the bandwidth in MB/s, the size in GB, and the time in seconds, which results in a combined metric unit that we call *adjusted GB seconds* (denoted AGBs). For the rest of this paper, we use this unit to express the cost.

C. Performance and cost prediction

Due to the massive amount of resources involved, it is important to anticipate the performance and cost of running a distributed scientific application at large scale, because this enables understanding whether it is feasible to obtain the desired results within a given time or cost budget.

However, in order for such an estimation to be possible, the application needs to exhibit a well defined I/O access pattern. Luckily, a majority of high-performance computing (HPC) numerical simulations exhibit such a well defined I/O access pattern: they are composed iterative bulk-synchronous computations consisting of a large number of distributed processes that simultaneously checkpoint their state to disk at regular intervals. We choose to illustrate our estimation proposal on this class of applications.

More specifically, we assume n VMs need to checkpoint the same amount of information at regular intervals. We denote the size of the checkpoint dumped by each VM as S_c. Furthermore, users typically know how many iterations I they need to run (e.g., number of timesteps in a simulation), how much compute time I_t is needed per iteration and how many checkpoints per VM (K) will be dumped during the application runtime (i.e., number of intervals). Even if S_c or I_t is not known in advance, such information can be obtained by running the application in benchmarking mode for a short period of time until at least one set of checkpoints can be captured and analyzed (i.e., $K = 1$). Assuming that the required information about the application was collected one way or another, the completion time T can be estimated as follows:

$$T = I \cdot I_t + \frac{K \cdot S_c}{B(S_c)}$$

In the above equation, $B(x)$ denotes the sustained throughput for writing a dataset of size x and is assumed to be known.

In addition to the completion time, we are interested in an estimation of the overall adjusted storage accumulation C_a, which is the sum of the individual accumulation C (introduced in Section III-B) of each VM. Since the checkpoint size S_c remains fixed throughout the application runtime, there is a size $M(S_c)$ of the caching device that corresponds to an optimal performance-cost trade-off. For the purpose of this section, we assume that $M(S_c)$ is given, while zooming in Section III-D on how to choose $M(S_c)$. Furthermore, in a typical scenario, the application performs an I/O initialization step where each VM dumps S_i GB worth of data corresponding to an initial state. Accounting for this additional initialization phase, C_a can be estimated as follows:

$$C = B_b \cdot N \cdot (T + \frac{S_i}{D(S_i)}) + B_c \cdot M(S_i) \cdot D(S_i) + K \cdot B_c \cdot M(S_c) \cdot D(S_c)$$

$$C_a = n \cdot C$$

In the above equation, $D(x)$ denotes the duration for which the caching device is attached while writing a dataset of size x and is assumed to be known.

Note that due to the complex interactions between the backing device and the caching device, it is not easy to determine $B(x)$ and $D(x)$ analytically. However, $B(x)$ and $D(x)$ depend on three parameters: throughput of the backing device B_b, throughput of the caching device B_c and size of the caching device M. Thus, we propose to establish $B(x)$ and $D(x)$ experimentally by selecting a series of representative values for x and M and running I/O intensive microbenchmarks (e.g. using *dd*) to directly measure $B(x)$ and $D(x)$. We refer to this process as *calibration*.

In a minimal configuration, two possible values for x (S_i and S_c) and M ($M(S_i)$ and $M(S_c)$) are enough to calibrate for a single scenario. However, the calibration can be extended to additional values for x and M in order to obtain multiple points that can be interpolated to estimate additional "what-if" scenarios useful for the application developer (e.g., the application developer may want to study what happens if the checkpoint size doubles as a consequence of increasing the floating point precision from single to double).

D. Optimal cache size selection based on performance and cost prediction

In the previous section we introduced a cost estimation technique based on a calibration phase that is able to anticipate the completion time and total adjusted storage accumulation for bulk-synchronous HPC applications that checkpoint at regular intervals. This was possible because of the assumption that the checkpoint size S_c remains constant, which leads to a situation where there optimal checkpointing cache size $M(S_c)$ also remains constant. In this section we introduce a technique that is able to recommend $M(S_c)$ for this particular scenario in advance, which can be used to enhance the generic adaptive cache sizing strategy introduced in Section III-A.

At first sight, a good choice seems $M(S_c) = S_c$, because it enables the cache to be large enough to hold all checkpointing data, which in turn enables the checkpointing to proceed at maximum speed. At the same time, it minimizes the adjusted storage accumulation required to do so, because the cache is not larger than absolutely necessary. However, this line of reasoning does not necessarily lead to an optimal configuration: if performance is not the top priority, then it could be beneficial to reduce $M(S_c)$ in order to obtain a lower adjusted storage accumulation. The opposite is also true: due to various overheads encountered in practice, it may be

beneficial to increase $M(S_c)$ beyond S_c in order to achieve the best throughput. Ultimately, both aspects introduce a level of complexity that makes it non-trivial to analytically determine an optimal $M(S_c)$.

To address this challenge, we propose a solution that is based on the cost estimation technique introduced in the previous section. The key idea behind our proposal is to perform an extended calibration phase that determines $B_M(x)$ and $D_M(x)$ for M in a range of values that facilitates an extended search around S_c according to the remarks mentioned above. Although this extended calibration may seem expensive, it is important to note that the microbenchmarks are independent of the users and applications, therefore cloud providers can cache the results to expose and reuse them later.

Once the results are available one way or another, T and C_a can be easily computed for all M. If there is a constraint on the maximum acceptable completion time (denoted T_{max}), then we keep only those $M(S_c)$ for which $T \leq T_{max}$. Subsequently, for the remaining $M(S_c)$, we compute the corresponding C_a and keep a single value that minimizes C_a, which is the recommended value.

E. Architecture

The simplified architecture of our approach is depicted in Figure 1. We assume that the VMs are deployed on an IaaS cloud that enables users to provision raw storage as virtual disks. Furthermore, we assume that the cloud hypervisor can dynamically attach and detach virtual disks to the VM instances (a standard feature in most production-ready hypervisors).

Once booted, the VM instance initializes an *adaptive block device* that uses a non-expensive virtual disk of limited throughput as the *backing device*. The adaptive block device is exposed inside the guest OS as a standard block device and can be leveraged as such (e.g., it can be formatted by using a file system). Once the adaptive block device is running, a *controller* daemon collects I/O statistics about it at fine granularity (e.g., sustained throughput); and, based on these statistics, it implements the design principles described in Section III-A. Specifically, during an I/O-intensive phase, it interacts with a standardized *IaaS API* in order to attach a virtual disk of optimized size (based on the experience from the previous I/O intensive phases) that will act as the *caching device*. Once the VM instance has recognized the caching device, it incorporates the corresponding guest-level block device into the adaptive block device. All interactions between the caching device and the backing device are handled transparently by our adaptive block device based on the principles mentioned in Section III-A. Once the I/O-intensive phase completes, the controller signals the adaptive block device to start flushing. After the flushing has completed, it detaches the corresponding virtual disk from the VM instance using the IaaS API and destroys the disk, thus stopping the accumulation of storage costs due to caching.

How to provision a virtual disk is open to a wide range of choices: various types of devices (e.g., HDDs, SSDs, RAM-disks) can be directly leveraged by the hypervisor and exposed as virtual disks inside the VM instance. If devices need to be shared, another option is to use virtual disk images of various formats (e.g., raw, QCOW2 [30]), hosted either locally or remotely on different types of devices. Virtual disks also may be provided by specialized services, such as Amazon EBS [1] or our own previous work [21]. Our approach is agnostic to any of these choices as long as they are handled through a standardized IaaS API.

F. Implementation details

In this section, we briefly introduce a prototype that implements the components presented in Section III-E.

We rely on *Bcache* [3] to implement the *adaptive block device*. Our choice was motivated by several factors. First, it offers out-of-the-box support to cache hot blocks of designated devices on other devices, while offering support to activate/deactivate caching in an online fashion. Second, it is highly configurable and offers detailed statistics about I/O utilization. In particular, the ability to control the caching strategy and the interaction between the backing device and caching device was crucial in enabling the implementation of the design principles presented in Section III-A. Third, it is implemented at the kernel level and is specifically designed to minimize performance overhead. Since we need to handle another level of indirection on top of the virtualized nature of the backing and caching device, this aspect is important in our context. Furthermore, it is part of the official Linux kernel and thus enjoys widespread exposure and adoption.

The *controller* was implemented as a Python daemon. We rely on *psutil* to get per-disk I/O statistics. The interaction with Bcache is implemented directly through the *sysfs* interface. We note also certain nontrivial aspects related to the management of virtual disks, in particular how to detect inside the guest when the disk is recognized by the kernel. To this end, we rely on *pyudev*, which implements an accessible interface to *libudev*, including asynchronous monitoring of devices in a dedicated background thread.

IV. Experimental evaluation

This section presents the experimental evaluation of our approach. We introduce the experimental setup and methodology (Section IV-A), and discuss results for microbenchmarks (Section IV-B) and two real-life HPC applications (Section IV-C and Section IV-D).

A. Experimental setup

Our experiments were performed on the *Shamrock* testbed of the Exascale Systems group of IBM Research in Dublin. For this work, we used a reservation of 32 nodes interconnected with Gigabit Ethernet, each of which features an Intel Xeon X5670 CPU (12 cores), HDD local storage of 1 TB, and 128 GB of RAM.

We simulate a cloud environment using *QEMU/KVM* 1.6 as the hypervisor. The VM instances run a recent Debian Sid (3.12 Linux kernel) as the guest operating system. The network

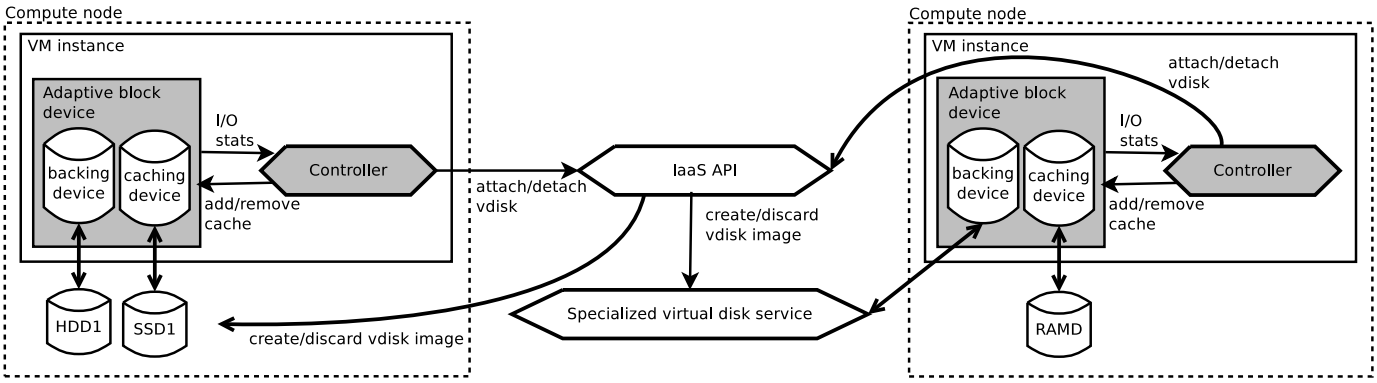


Fig. 1. Integration of our approach into an IaaS cloud architecture; components that are part of our design are highlighted with a darker background.

interface of each VM uses the *virtio* driver and is bridged on the host with the physical interface in order to enable point-to-point communication between any pair of VMs.

We compare three approaches throughout our evaluation:

Static preallocation using a slow virtual disk: In this setting, a large, fixed-size virtual disk is created on the local HDD of each host as a RAW file and attached to each VM instance after booting. The maximum I/O bandwidth of the virtio driver is fixed by using the hypervisor monitor, and the host-side caching is disabled to avoid interference. After booting the VM instances, all virtual disks are formatted by using the *ext4* file system, and the corresponding mount points are used for all I/O generated during the experiments. We refer to this setting as *static-slow*.

Static preallocation using a fast virtual disk: This setting is similar to the one described above, except that the fixed-sized virtual disk is hosted as a RAW file in a RAM-disk. Again, the maximum bandwidth available to the guest operating system is fixed. However, it is several times higher than in the previous case and is intended to simulate a faster device, such as an SSD. We refer to this setting as *static-fast*.

a) Transparent throughput elasticity using our approach: In this setting, we use a virtual disk with properties identical to those in the *static-slow* case as a backing device. Whenever more bandwidth is needed, a new virtual disk with properties identical to those in the *static-fast* case is used as a caching device to temporarily boost I/O throughput. The size of the caching device, as well as the moment when to attach and detach it, is automatically determined by our approach during runtime (as explained in Section III-A). Furthermore, the utilization threshold UT is set to 30%, and the inactivity delay ID is set to 30s. We refer to this setting as *adaptive*.

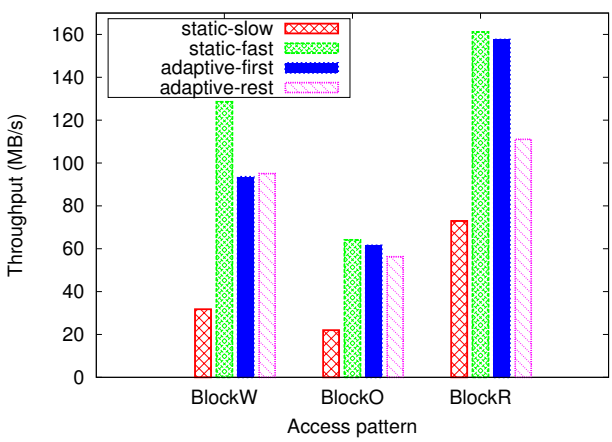
These approaches are compared based on the following metrics:

- *Performance overhead* is the difference in performance observed between *static-fast*, which is used as a baseline for the best possible performance, and the other two approaches that leverage slower virtual disks. In the case of microbenchmarks, performance refers to the sustained I/O throughput as perceived by the application. In the case of real-life applications, performance refers to the completion time, which measures the overall end-impact of each approach on the application runtime.
- *Total adjusted storage accumulation* is the sum of the adjusted storage accumulation for all VM instances involved in the experiment. It is used to quantify storage-related costs for the entire application deployment according to the cost model introduced in Section III-B.
- *Evolution of I/O activity* represents the total I/O bandwidth utilization (due to reads and writes to virtual disks) for all VM instances. In the *static-fast* and *static-slow* cases, it overlaps with the sustained I/O throughput as perceived by the application. In the *adaptive* case, it measures all background I/O activity to the backing device and caching device (which can be higher than *static-fast* when both devices are active simultaneously). This metric is important for studying how the compute phases interleave with the I/O phases and how the backing device interacts with the caching device during this interleaving.

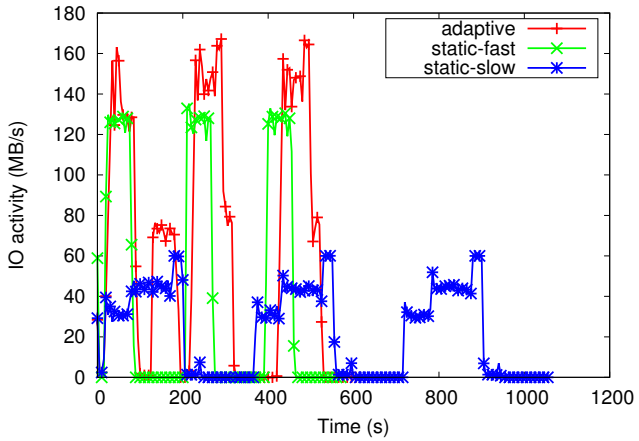
B. Microbenchmarks

Our first series of experiments focuses on the I/O performance of all three approaches in synthetic settings. For this purpose, we use *Bonnie++*, a standard I/O benchmarking tool that measures read, write, and rewrite throughput when using 32 KB blocks (default value used in the experiments). We focus on these values because they are the most representative of real-life large-scale scientific applications (as opposed to byte-by-byte read/write throughput or other file-system related statistics that are reported).

We run the following experiment: a single VM instance is booted and Bonnie++ is launched three times in a row, with a 120 second pause between each run. We repeat the experiment for each approach three times and record the average of the relevant Bonnie++ statistics. The memory allocated to the VM is fixed at 2 GB. The virtual disk settings are as follows: the size of the backing device is fixed at 10 GB, with a reserved bandwidth of 60 MB/s for both *static-slow* and *adaptive* and a reserved bandwidth of 128 MB/s for *static-fast*. For *adaptive*, the reserved bandwidth of the caching device is fixed at 128 MB/s.



(a) Throughput for block-write (BlockW), block-overwrite (BlockO), and block-read (BlockR)



(b) I/O activity (reads and writes from the backing device and, if applicable, caching device)

Fig. 2. Bonnie++: I/O performance under different access patterns and its corresponding I/O activity

The Bonnie++ statistics are depicted in Figure 2(a). Since our approach automatically adjusts the size of the caching device after the first iteration, we depict the results for the first run and the remaining two runs separately. The initial cache size is set to 10 GB, which is lowered by our approach to 6 GB for the consecutive iterations. We denote the first iteration *adaptive-first*, and the average of the second and third iteration *adaptive-rest*.

As can be observed, in the case of *static-slow*, the backing HDD limits the write throughput to 33 MB/s (out of 60 MB/s), which is not the case for *static-fast*, where the maximum write throughput of 128 MB/s can be achieved). At approximately 100 MB/s, both adaptive approaches have an overhead of 30% of write throughput compared with *static-fast*. With respect to overwrite throughput, all approaches suffer performance degradation compared with a simple write, which is due to the read, seek back, and write cycle employed by Bonnie++ for each block. The large gap between *static-slow* and the rest is still present; however, this time the overhead between the two adaptive approaches and *static-fast* is almost negligible. With respect to read throughput, a considerable increase is present for all approaches as a result of caching. Also, for the first time, a visible difference is noticeable between the two adaptive approaches: *adaptive-first* is comparable to *static-fast* whereas *adaptive-rest* has an overhead 25%. This overhead is due to the smaller size of the caching device, which forces writeback to the backing device earlier (and thus causes reads from the caching device).

To understand the interaction between the backing device and the caching device better, we depict the evolution of I/O activity (as measured at five-second granularity) in Figure 2(b). As expected, both static approaches have a highly deterministic behavior, with a flatter and elongated pattern observable for *static-slow*. In the case of *adaptive*, an initial burst is observable for the first run, which is followed by a smaller secondary burst. This secondary burst corresponds to cache flushing after ID elapsed that triggers the detach request. Thanks to the automatic adjustment of the size of the caching device to a smaller value, we observe earlier writeback. This causes a shorter flush burst that is fused into the primary burst, effectively enabling the caching device to be detached sooner (which reduces utilization cost).

C. Case study: CM1

Our next series of experiments evaluates the behavior of our system for *CM1*, a real-life HPC application that is a three-dimensional, non-hydrostatic, nonlinear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. This MPI application is used to study small-scale processes that occur in the atmosphere of the Earth (such as hurricanes) and is representative of a large class of HPC bulk-synchronous stencil applications that exhibit an iterative behavior of alternating between a compute phase and an I/O-intensive phase to write intermediate output results and checkpoints (used to restart from in case of failures).

For this work, we have chosen as input data a 3D hurricane that is a version of the Bryan and Rotunno simulations [4]. We run the simulation of this 3D hurricane on 32 VMs, with each VM hosted on a separate physical node and equipped with 11 virtual cores (out of which 10 are reserved for CM1 and 1 reserved for guest OS overhead). Furthermore, 1 core is reserved for the hypervisor. Each VM is allocated 18 GB of RAM, enough to fill the need of the MPI processes. Thus, the overall setup totals 320 MPI processes that generate a heavy load on the underlying nodes. The output/checkpointing frequency is set at 50 simulation time steps, out of a total of 160 time steps. This setup leads to the following access pattern: right after initialization, the application dumps the initial state, which causes a first I/O-intensive phase. After that there follow three more I/O-intensive phases of higher magnitude that are interleaved with computational phases.

The settings are as follows: the size of the backing device is fixed at 50 GB, while the reserved bandwidth is fixed at 33 MB/s for *static-slow* and *adaptive* (according the maximum write throughput observed in microbenchmarks in order to avoid over-provisioning). The bandwidth in the case of *static-fast* is fixed at 128 MB/s. The caching device has a bandwidth of 128 MB/s and an initial size of 10 GB. Each experiment is repeated five times for all three approaches, and the results are averaged.

Performance results are summarized in Table I. As can be observed, speeding the I/O phase can lead to a significant boost in overall completion time: compared with *static-fast*, which is used as a baseline, *adaptive* has a small overhead of 3.3%, which contrasts with the large overhead of 23% observed for *static-slow*. These results are significant both directly (the users want a minimal time to solution) and indirectly (longer runtimes mean the VMs need to stay up longer and thus generate more costs).

TABLE I
CM1 (NUMERICAL MODEL DESIGNED FOR IDEALIZED STUDIES OF ATMOSPHERIC PHENOMENA): PERFORMANCE RESULTS

Approach	Completion Time	Overhead
static-slow	1471s	23%
static-fast	1190s	–
adaptive	1231s	3.3%

Even if the overhead of *adaptive* is small, it is justifiable only if the storage space and bandwidth utilization can be significantly lowered according to the cost model introduced in Section III-B. To quantify these costs, we compute the adjusted storage accumulation for all three approaches. More specifically, for each VM instance i, in the case of *static-slow* we have $C_i(t) = 33 \cdot 50 \cdot t$, while for *static-fast* we have $C_i(t) = 128 \times 50 \cdot t$. For *adaptive*, we have

$$C_i(t) = 33 \cdot 50 \cdot t + 128 \cdot \int_0^t M_i(x) \cdot dx$$

$M_i(x)$ is the size of the caching device for VM instance i at moment x and is automatically determined by our approach (0 if the caching device is not in use). To facilitate the calculation of $C_i(t)$ in practice, we assume a discretization of time at five-second granularity (i.e., we probe for the value $M_i(x)$ every five seconds and assume it stays constant during this interval). Since we have a total of 32 VM instances, the total cost in each of the three cases is

$$C_a(t) = \sum_{i=1}^{32} C_i(t)$$

C_a is expressed in AGBs (introduced in Section III-B) and is depicted in Figure 3(a). One can see a large gap between *static-slow* and *static-fast*, which steadily grows as the application progresses in time. However, not only is *adaptive* very close to *static-slow*, but overall it even manages to reduce the total adjusted storage accumulation by almost 7% because the application finishes faster. Compared with *static-fast*, this amounts to a reduction of 65% in cost, which is a large gain for the price of 3.3% performance overhead.

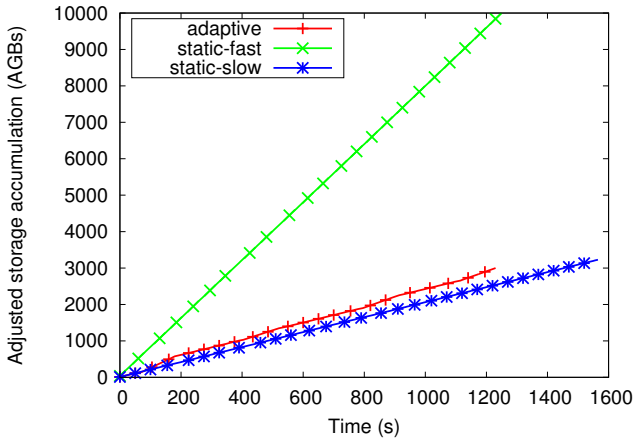
The evolution of total I/O activity in depicted in Figure 3(b). To calculate it, we divide the time (x axis) in five-second intervals and sum the I/O throughput (expressed in MB/s) observed during each interval for all VM instances (y axis). Since the application is bulk-synchronous, the I/O-intensive phases and the compute intensive phases overlap for all VM instances at the same time, leading to a clear delimitation in terms of I/O activity for all three approaches.

In the case of *static-fast* and *static-slow*, the I/O phases are short but of high amplitude and, respectively, longer but of lower amplitude. These results directly correspond to the ability of the underlying backing device to handle a fixed amount of I/O activity using a high and, respectively, low reserved bandwidth. With our approach, the behavior is more complex: the first I/O phase of the application generates a high I/O burst, followed by a clearly visible flush period. Since the first I/O phase is less intensive than the rest (for all three approaches), our approach picks a 4 GB cache device for the second phase, which then is increased to 6 GB for the last two phases. All I/O phases except the first exhibit an I/O burst that is fused to the flush period, with only a small difference noticeable between the second phase and the third, which itself is almost identical to the fourth. Intuitively, this hints at the ability of our approach to optimally adjust the cache size based on the previous I/O-intensive phases.

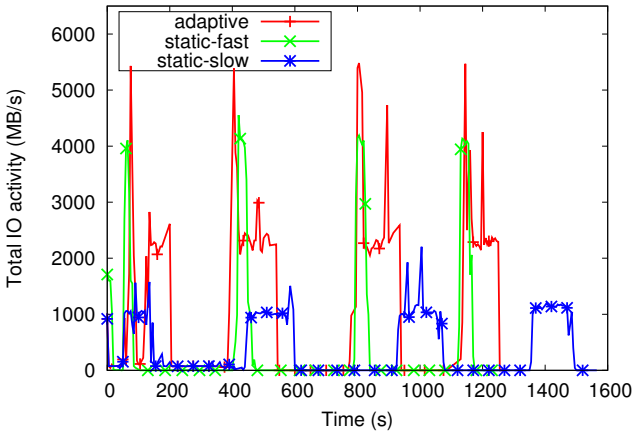
D. Case study: LAMMPS

A second real-life HPC application we use to demonstrate the benefits of our approach is *LAMMPS* [29], a large-scale atomic/molecular massively parallel simulator. LAMMPS can be used to model atoms or, more generically, particles at the atomic, meso, or continuum scale. Such modeling is useful in understanding and designing solid-state materials (metals, semiconductors), soft matter (biomolecules, polymers), and coarse-grained or mesoscopic systems. Similar to CM1, it exhibits an iterative behavior of alternating between a compute phase and an I/O-intensive phase to write intermediate output results and checkpoints, which are used for restart in case of failures.

For this work, we have chosen as input data a 3D Lennard-Jones melting scenario. Melting, the phenomenon of phase transition from a crystalline solid state to a liquid state, is one of the most important phase transformations in the processing and applications of materials, playing an important role in materials science and engineering [9]. As in the case of CM1, we run the simulation on 32 VMs, with each VM hosted on a separate physical node and equipped with 11 virtual cores, each of which corresponds to a physical core, with the remaining physical core reserved for the hypervisor. Inside each VM, 10 of the virtual cores are reserved for MPI processes, while the remaining core is reserved for operating system overhead. Each MPI process is responsible for a 20x160x160 subdomain. Thus, the total deployment amounts to 320 MPI processes that are evenly spread over 32 VMs



(a) Evolution of total adjusted storage accumulation for all 32 VM instances (lower is better)



(b) Total I/O activity (reads and writes from all backing devices and, if applicable, caching devices of all 32 VM instances)

Fig. 3. CM1 (numerical model designed for idealized studies of atmospheric phenomena): a real-life HPC application that runs on 32 VMs (each on a different node) using 10 MPI processes/node

and process a 6400x160x160 grid. The output/checkpointing frequency is set at 30 simulation time steps, out of a total of 100 time steps. This results in an initial I/O- intensive phase to dump the initial state, followed by three I/O- intensive phases interleaved with computational phases.

The settings are identical to the setup used in Section IV-C for CM1: the size of the backing device is fixed at 50 GB, with a reserved bandwidth of 33 MB/s for *static-slow* and *adaptive*. The bandwidth of the backing device is fixed at 128 MB/s for *static-fast*. The caching device has a bandwidth of 128 MB/s and an initial size of 10 GB. Each experiment is repeated five times for all three approaches, and the results are averaged.

Performance results are depicted in Table II, where *static-fast* is used as a baseline for the fastest possible completion time. As can be observed, using a slow backing device in the case of *static-slow* leads to a significant increase (15.5%) in overall completion time when compared with *static-fast* because of the longer I/O phases. On the other hand, *adaptive* successfully reduces the overhead of the I/O phases to such extent that the overall increase in completion time becomes negligible (i.e., around 1%).

TABLE II
LAMMPS (LARGE-SCALE ATOMIC/MOLECULAR MASSIVELY PARALLEL SIMULATOR): PERFORMANCE RESULTS

Approach	Completion Time	Overhead
static-slow	1680s	15.5%
static-fast	1454s	–
adaptive	1476s	1.01%

For this negligible increase in completion time, our approach achieves massive reductions in cost when compared with *static-fast* and even a moderate reduction in cost when compared to *static-slow*. These results are depicted in Figure 4(a) as the total adjusted storage accumulation for all instances ($C_a(t)$). For all three approaches, $C_a(t)$ is calculated in the same way as is described in Section IV-C (since we use the same configuration as in the case of CM1).

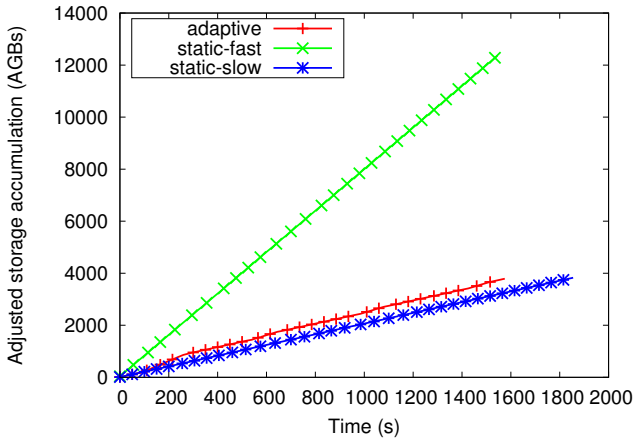
More specifically, *adaptive* reduces the storage accumulation by more than 70% when compared with *static-fast* and marginally by 1% compared with *static-slow* (thanks to the fact that it finishes faster), effectively making it a double winner over *static-slow* in terms of both performance and cost.

To understand how the interaction between backing devices and caching devices contributes to the results, we again analyze the total I/O activity. In Figure 4(b) a pattern similar to the case of CM1 is observable: *static-fast* and *static-slow* exhibit a short but intense I/O burst vs. a longer but milder I/O burst, with our approach successfully attaching the caching device at the right time and adapting its size based on the past experience (demonstrated by the shorter and more regular flush periods after the initial burst for the last three I/O-intensive phases when compared with the initial I/O-intensive phase where the optimal cache size is unknown).

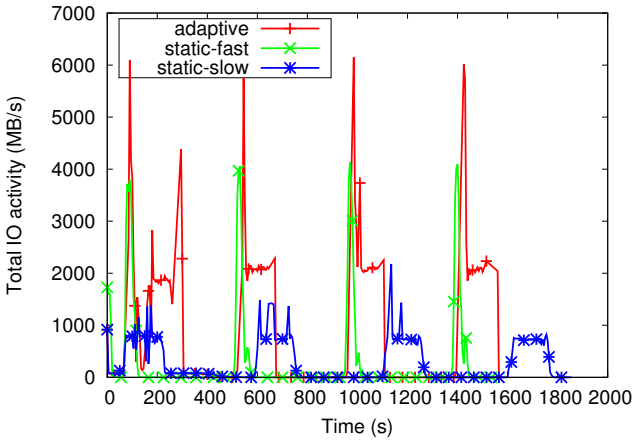
E. Prediction and optimization of performance and cost

In this section, we apply the techniques presented in Section III-C and Section III-D to the two HPC applications introduced above: CM1 and LAMMPS. We aim to understand: (1) how accurately our estimation is compared to the results observed in real life; (2) what cache size is recommended by our optimization technique and what impact this would have.

As a first step, we perform an extended calibration using *dd* to write a variable dataset size (3 GB, 4 GB, 6 GB and 9 GB) in chunks of 1 MB. These dataset sizes match the write patterns observed for the applications: in the case of CM1, there is an initial dump of state amounting to $S_i = 5\ GB$, while the checkpoint size $S_c = 4\ GB$. In the case of LAMMPS, $S_i = 4\ GB$, while $S_c = 3\ GB$. The cache size ranges from 2 GB to 12 GB, using an increment of 2 GB. Figure 5 depicts the results of the calibration in terms of sustained throughput

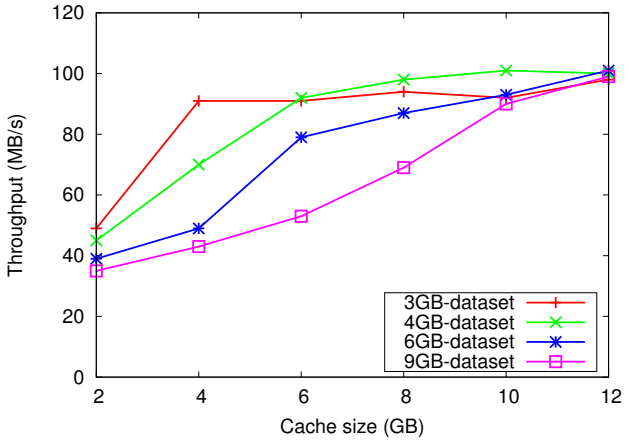


(a) Evolution of total adjusted storage accumulation for all 32 VM instances (lower is better)

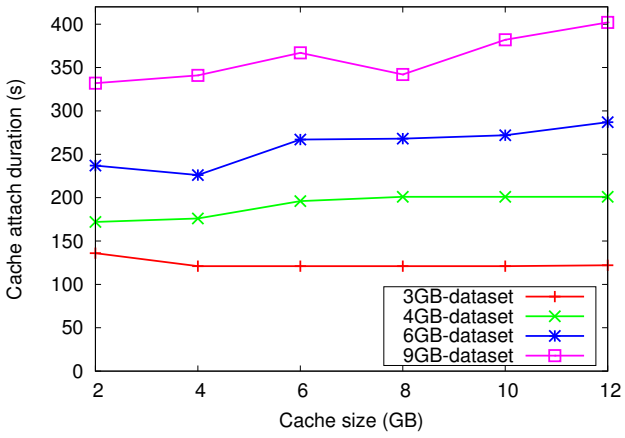


(b) Total I/O activity (reads and writes from all backing devices and, if applicable, caching devices of all 32 VM instances)

Fig. 4. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator): a real-life HPC application that runs on 32 VMs (each on a different node) using 10 MPI processes/node



(a) Sustained throughput using *dd* to write a fixed-sized dataset (higher is better)



(b) Amount of time for which the caching device stays attached (lower is better)

Fig. 5. Calibration phase: Sustained throughput and cache attach duration using a variable cache size.

(Figure 5(a)) and for how long the cache remained attached (Figure 5(b)).

With respect to the sustained throughput, there are several important observations. First, the maximum sustained throughput (i.e., 100 MB/s) is below the maximum bandwidth of the caching device (i.e., 128 MB/s), which is expected because there is a delay between the moment when the I/O intensive phase starts and the moment when caching device gets attached and can accelerate the *dd* writes. Second, as anticipated in Section III-D, it can be observed that the peak throughput is achieved when the cache size is larger than the dataset size, due to the extra overhead in managing two block devices simultaneously. Interesting to note is that the increase continues well beyond the dataset size. Conversely, when the cache size is smaller than the dataset size, the sustained throughput drops considerably. Analyzing the duration for which the caching device remains attached, it can be observed that the cache size has much less impact than the dataset size. This result is expected, as the caching device needs to stay attached only for as long as it is necessary to accumulate the written data and then flush it, which is not dependent on cache size.

Using these measurements, our first goal is to assess how accurate the estimation of the completion time and adjusted storage accumulation is when compared with the real-life results presented in Section IV-C and Section IV-D. To this end, we first measure the baseline (i.e. completion time when running without any checkpointing), which corresponds to $I \cdot I_t$: for CM1 we obtained 1100s, while for LAMMPS we obtained 1380s. Then, using the equations for T and C_a

(introduced in Section III-C), we compute the corresponding estimations for all three approaches: for *static-fast* and *static-slow*, we use $B_c = 0$ and fix $B(x) = 128$ and, respectively, $B(x) = 33$. For *adaptive*, the initial cache size is $M(S_i) = 10\ GB$ (matching the setting used for the real-life experiments), while $M(S_c) = S_c + 2\ GB$, which is the dynamic cache size selected by the access pattern aware cache sizing during the checkpointing phase. Since there are three checkpoints, $K = 3$. The estimations for T and C_a are then normalized to their real-life counterparts to obtain the accuracy. The results are summarized in Table III and, respectively, Table IV. As can be observed, the completion time can be predicted with high accuracy: the error is at most 1.3% underestimation, with virtually no overestimation (i.e. less than 0.5%). On the other hand, the total adjusted storage accumulation is harder to predict with the same level of accuracy: the maximum error ranges from a 4% underestimation to a 10% overestimation.

TABLE III
ACCURACY OF PERFORMANCE PREDICTION: ESTIMATED OVER REAL COMPLETION TIME

Approach	CM1	LAMMPS
static-slow	1.0009270132	0.9876623377
static-fast	1.0050420168	0.9986244842
adaptive	1.0020838484	1.0035736621

TABLE IV
ACCURACY OF COST PREDICTION: ESTIMATED OVER REAL TOTAL ADJUSTED STORAGE ACCUMULATION

Approach	CM1	LAMMPS
static-slow	1.0399457837	0.9613944238
static-fast	0.9888	0.9605177994
adaptive	1.1063781856	0.9898244917

For the reminder of this section, we proceed with a study of how the selection of the cache size during the checkpointing phase impacts the estimated completion time and total adjusted storage accumulation. This illustrates how our proposal introduced in Section III-D can be used to optimize the access pattern aware cache sizing strategy in order to achieve a minimal cost for a given maximum acceptable completion time. To this end, we use the same parameters as before, but vary $M(S_c) \in [2..12]$ in the calculation of T and C_a for *adaptive*, which are then normalized against their estimated (fixed) counterparts from *static-fast*. The results are depicted in Figure 6(a) and Figure 6(b).

As expected, a higher cache size leads to a smaller estimated degradation of performance compared with *static-fast*. However, the benefits for using a $M(S_c)$ larger than 6 GB for CM1 and 4 GB for LAMMPS are becoming marginal. On the other hand, the estimated storage accumulation is minimized by fixing $M(S_c) = 4\ GB$ in both cases. Thus, for LAMMPS the minimum cost and performance degradation coincide, whereas for CM1 it is not possible to aim for the minimum cost unless a performance degradation of 7% is acceptable. In both cases, using $M(S_c)$ less than 4 GB leads to severe degradation of performance, which also raises the cost (because of longer runtime).

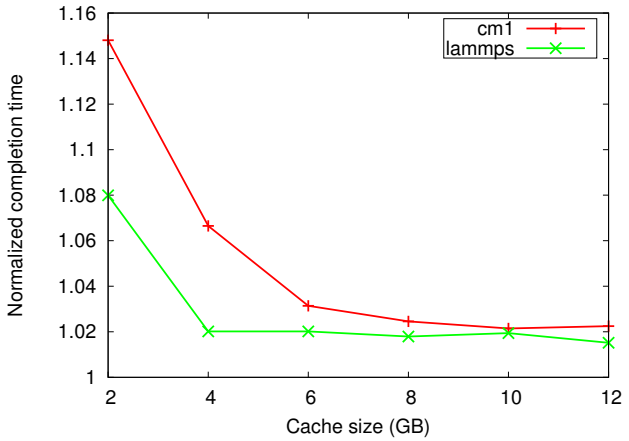
V. CONCLUSIONS

The ability to transparently adapt to the fluctuations of I/O throughput requirements in order to reduce the costs associated with over-provisioning faster (and more expensive) virtual disks is crucial in the context of large-scale scientific applications for two reasons: (1) such applications tend to have an iterative behavior that interleaves computational phases with I/O-intensive phases, which leads to extreme fluctuations in I/O requirements that make over-provisioning particularly wasteful; and (2) such applications run in configurations that include a large number of VMs and associated virtual disks, which amplifies the waste due to scale.

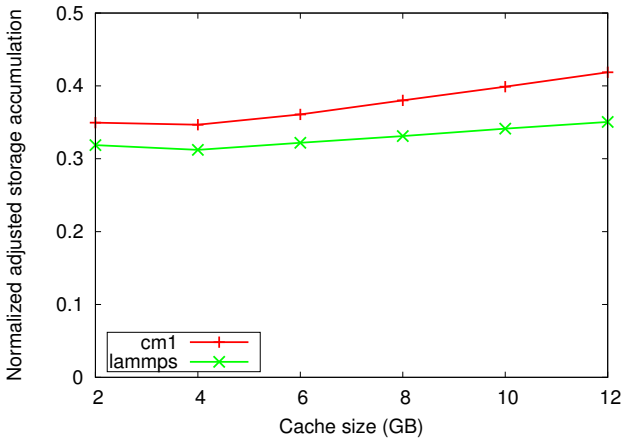
In this paper we have described a solution that relies on a specialized block device exposed inside the guest operating system in order to intercept all I/O to a potentially slow virtual disk and enhance its throughput during I/O peaks by leveraging an additional faster, short-lived virtual disk that acts as a caching layer. Our approach offers low-level transparency that enables any higher-level storage abstraction to benefit from throughput elasticity, including those not originally designed to run on IaaS clouds. Furthermore, it relies solely on standard virtual disks to operate, which offers a high degree of compatibility with a wide range of cloud providers.

We demonstrated the benefits of this approach through experiments that involve dozens of nodes, using both microbenchmarks and two representative real-life HPC applications: *CM1* and *LAMMPS*. Compared with static approaches that over-provision fast virtual disks to accommodate the I/O peaks, our approach demonstrates a reduction of storage cost in real life (using a cost model that charges users proportionally to disk size and reserved bandwidth) of 66%–70%, all of which is possible with a negligible performance overhead (1%–3.3%) when compared with the fastest and more expensive solution. Furthermore, our findings show that using slow virtual disks to minimize storage costs is not the optimal solution: because of higher performance degradation (15%–23%), applications end up using the cheap virtual disks for longer, thereby resulting in an overall increase in cost compared with our approach (1%–7%).

Furthermore, we have introduced a performance and cost prediction model specifically for HPC applications that interleave computationally intensive phases with write intensive phases corresponding to checkpointing. In this case, our approach aims to estimate the completion time and total adjusted storage accumulation. We have validated our proposal for the two real-life HPC application case studies: compared with the real-life results, our estimations have a maximal error of 1.3% for completion time and 10% for adjusted storage accumulation, with a best case scenario of less than 0.5% error otherwise. We also introduced a methodology to optimize the cache size used during checkpointing in order to achieve minimal cost depending on maximum acceptable completion time, which complements our the generic access pattern aware



(a) Estimated completion time, normalized to *static-fast* (lower is better)



(b) Estimated storage accumulation, normalized to *static-fast* (lower is better)

Fig. 6. Predicted impact of cache size on performance and cost

cache sizing strategy. This approach was illustrated for both real-life HPC applications.

Encouraged by these initial results, we plan to develop this work in several directions. One straightforward extension is to explore how to leverage multiple virtual disks, potentially in striping configuration, in order to boost I/O throughput during peaks. Another interesting idea could be to explore putting the backing device itself in stand-by during compute-intensive phases and to use a smaller caching device of equal bandwidth capability to serve most I/O requests, under the assumption that it makes sense to pay for the penalty of reattaching the backing device in order to deal with cache misses and evictions. Such an approach has the potential to further reduce the costs. Also, we assumed a cost model that charges solely on reserved throughput, without considering the amount and frequency of I/O operations. Thus, another interesting direction is to explore an extended cost models where both aspects are considered.

ACKNOWLEDGMENTS

This material is based in part on work supported in part by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

REFERENCES

[1] Amazon Elastic Block Storage (EBS). http://aws.amazon.com/ebs/.

[2] Amazon Elastic Block Storage (S3). http://aws.amazon.com/s3/.

[3] BCache. http://bcache.evilpiepirate.org.

[4] George H. Bryan and Richard Rotunno. The maximum intensity of tropical cyclones in axisymmetric numerical model simulations. *Journal of the American Meteorological Society*, 137:1770–1789, 2009.

[5] Yuan-Hao Chang, Ping-Yi Hsu, Yung-Feng Lu, and Tei-Wei Kuo. A driver-layer caching policy for removable storage devices. *Trans. Storage*, 7(1):1:1–1:23, June 2011.

[6] Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. Walnut: A unified cloud object store. In *SIGMOD '12: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 743–754, Scottsdale, USA, 2012. ACM.

[7] David Chiu, Apeksha Shetty, and Gagan Agrawal. Elastic cloud caches for accelerating service-oriented computations. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, New Orleans, USA, 2010. IEEE Computer Society.

[8] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[9] ChandanK. Das and JayantK. Singh. Melting transition of confined lennard-jones solids in slit pores. *Theoretical Chemistry Accounts*, 132(4), 2013.

[10] Brian C. Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *FAST'02: Proc. 1st USENIX Conference on File and Storage Technologies*, pages 5–5, 2002.

[11] Google. Google Compute Engine Pricing. https://developers.google.com/compute/pricing, 2014.

[12] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *SC '10: Proceedings of the 23rd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, New Orleans, USA, 2010. IEEE Computer Society.

[13] Heeseung Jo, Youngjin Kwon, Hwanju Kim, Euiseong Seo, Joonwon Lee, and Seungryoul Maeng. Ssd-hdd-hybrid virtual disk in consolidated environments. In *Euro-Par'09: Proc. 15th International Conference on Parallel Processing*, pages 375–384, Delft, The Netherlands, 2010.

[14] Zisis Karampaglis, Anastasios Gounaris, and Yannis Manolopoulos. A bi-objective cost model for database queries in a multi-cloud environment. In *MEDES '14: The 6th International Conference on Management of Emergent Digital EcoSystems*, pages 19:109–19:116, Buraidah, Al Qassim, Saudi Arabia, 2014.

[15] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *ICAC '10: Proc. 7th International Conference on Autonomic computing*, pages 1–10, Washington DC, USA, 2010.

[16] Mingliang Liu, Ye Jin, Jidong Zhai, Yan Zhai, Qianqian Shi, Xiaosong Ma, and Wenguang Chen. Acic: Automatic cloud i/o configurator for hpc applications. In *SC '13: The 26th International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 38:1–38:12, Denver, Colorado, 2013.

[17] Tian Luo, Siyuan Ma, Rubao Lee, Xiaodong Zhang, Deng Liu, and Li Zhou. S-cave: Effective ssd caching to improve virtual machine storage performance. In *PACT '13: Proc. 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 103–112, Edinburgh, UK, 2013.

[18] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and

meet application deadlines in cloud workflows. In *SC '11: Proc. 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 49:1–49:12, Seattle, USA, 2011.

[19] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *CCGRID '10: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52, Melbourne, Australia, 2010.

[20] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(4):41–54, April 2008.

[21] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going back and forth: Efficient multi-deployment and multi-snapshotting on clouds. In *HPDC '11: 20th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 147–158, San José, USA, 2011.

[22] Bogdan Nicolae and Franck Cappello. BlobCR: Virtual disk based checkpoint-restart for HPC applications on IaaS clouds. *Journal of Parallel and Distributed Computing*, 73(5):698–711, May 2013.

[23] Bogdan Nicolae, Alexei Karve, and Andrzej Kochut. Discovering and Leveraging Content Similarity to Optimize Collective On-Demand Data Access to IaaS Cloud Storage. In *CCGrid'15: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 211–220, Shenzhen, China, 2015.

[24] Bogdan Nicolae and Mustafa Rafique. Leveraging Collaborative Content Exchange for On-Demand VM Multi-Deployments in IaaS Clouds. In *Euro-Par '13: 19th International Euro-Par Conference on Parallel Processing*, pages 305–316, Aachen, Germany, 2013.

[25] Bogdan Nicolae, Pierre Riteau, and Kate Keahey. Bursting the cloud data bubble: Towards transparent storage elasticity in iaas clouds. In *IPDPS '14: Proc. 28th IEEE International Parallel and Distributed Processing Symposium*, Phoenix, USA, 2014.

[26] Bogdan Nicolae, Pierre Riteau, and Kate Keahey. Transparent Throughput Elasticity for IaaS Cloud Storage Using Guest-Side Block-Level Caching. In *UCC'14: 7th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 186–195, London, UK, 2014.

[27] Shuangcheng Niu, Jidong Zhai, Xiaosong Ma, Xiongchao Tang, and Wenguang Chen. Cost-effective cloud hpc resource provisioning by building semi-elastic virtual clusters. In *SC' 13: Proc. 26th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 56:1–56:12, Denver, USA, 2013.

[28] Diego Perez-Palacin, Radu Calinescu, and José Merseguer. Log2cloud: Log-based prediction of cost-performance trade-offs for cloud deployments. In *SAC '13: The 28th Annual ACM Symposium on Applied Computing*, pages 397–404, Coimbra, Portugal, 2013.

[29] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117(1):1–19, March 1995.

[30] The QCOW2 Image Format. https://people.gnome.org/~markmc/qcow-image-format.html.

[31] Sheng Qiu and A. L. Narasimha Reddy. Nvmfs: A hybrid file system for improving random write in nand-flash ssd. In *MSST' 13: Proc. 38th International Conference on Massive Storage Systems and Technology*, pages 1–5, Lake Arrowhead, USA, 2013.

[32] SoftLayer. http://www.softlayer.com/.

[33] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. Logbase: A scalable log-structured database system in the cloud. *Proc. VLDB Endow.*, 5(10):1004–1015, June 2012.

[34] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *Trans. Storage*, 2(3):309–348, August 2006.

[35] Yang Wang, Jiwu Shu, Guangyan Zhang, Wei Xue, and Weimin Zheng. Sopa: Selecting the optimal caching policy adaptively. *Trans. Storage*, 6(2):7:1–7:18, July 2010.

[36] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[37] Guanying Wu, Xubin He, and Ben Eckart. An adaptive write buffer management scheme for flash-based ssds. *Trans. Storage*, 8(1):1:1–1:24, February 2012.

[38] Yiqi Xu, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato J. O. Figueiredo, and Seetharami Seelam. vPFS: Bandwidth virtualization of parallel storage systems. In *MSST' 12: Proc. 38th International Conference on Massive Storage Systems and Technology*, pages 1–12, Pacific Grove, USA, 2012.